PARCOACH Extension for Hybrid Applications with Interprocedural Analysis

Emmanuelle Saillard, Hugo Brunie, Patrick Carribault, Denis Barthou

PARCOACH Extension for Hybrid Applications with Interprocedural Analysis

Emmanuelle Saillard, Hugo Brunie, Patrick Carribault and Denis Barthou

Abstract Supercomputers are rapidly evolving with now millions of processing units, posing the questions of their programmability. Despite the emergence of more widespread and functional programming models, developing correct and effective parallel applications still remains a complex task. Although debugging solutions have emerged to address this issue, they often come with restrictions. Furthermore, programming model evolutions stress the requirement for a validation tool able to handle hybrid applications. Indeed, as current scientific applications mainly rely on MPI (Message-Passing Interface), new hardwares designed with a larger node-level parallelism advocate for an MPI+X solution with X a shared-memory model like OpenMP. But integrating two different approaches inside the same application can be error-prone leading to complex bugs. In an MPI+X program, not only the correctness of MPI should be ensured but also its interactions with the multi-threaded model. For example, identical MPI collective operations cannot be performed by multiple non-synchronized threads. In this paper, we present an extension of the PARallel COntrol flow Anomaly CHecker (PARCOACH) to enable verification of hybrid HPC applications. Relying on a GCC plugin that combines static and dynamic analysis, the first pass statically verifies the thread level required by an MPI+OpenMP application and outlines execution paths leading to potential deadlocks. Based on this analysis, the code is selectively instrumented, displaying an error and interrupting all processes if the actual scheduling leads to a deadlock situation.

Emmanuelle Saillard
CEA, DAM, DIF, F-91297 Arpajon, France e-mail: emmanuelle.saillard.ocre@cea.fr

Hugo Brunie
CEA, DAM, DIF, F-91297 Arpajon, France e-mail: hugo.brunie.ocre@cea.fr

Patrick Carribault
CEA, DAM, DIF, F-91297 Arpajon, France e-mail: patrick.carribault@cea.fr

Denis Barthou
Bordeaux Institute of Technology, LaBRI / INRIA, Bordeaux, France e-mail: denis.barthou@labri.fr

1 Introduction

The evolution of supercomputers to Exascale systems raises the issue of choosing the right parallel programming models for applications. Currently, most HPC applications are based on MPI. But the hardware evolution of increasing core counts per node leads to a mix of MPI with shared-memory approaches like OpenMP. However merging two parallel programming models within the same application requires full interoperability between these models and makes the debugging task more challenging. Therefore, there is a need for tools able to identify functional bugs as early as possible during the development cycle. To tackle this issue, we designed the PARallel COntrol flow Anomaly CHecker (PARCOACH) that combines static and dynamic analyses to enable an early detection of bugs in parallel applications. With the help of a compiler pass, PARCOACH can extract potential parallel deadlocks related to control-flow divergence and issue warnings during the compilation. Not only the parallel contructs involved in the deadlock are identified and printed during the compilation, but the statements responsible for the control-flow divergence are also outputed. In this paper, we propose an extension of PARCOACH to hybrid MPI+OpenMP applications and an interprocedural analysis to improve the bug detection through a whole program. This work is based on [9] and extends [10] with more details and an interprocedural analysis. To the best of our knowledge, only Marmot [3] is able to detect errors in MPI+OpenMP programs. But as a dynamic tool, Marmot detects errors during the execution and is limited to the dynamic parallel schedule and only detects errors occuring for a given inputset whereas our approach allows for static bug detection with runtime support and detects bugs for all possible values of inputs.

In the following we assume that all programs are SPMD MPI programs and all MPI collective operations are called with compatible arguments (only the `MPI_COMM_WORLD` communicator is supported). Therefore, each MPI task can have a different control flow within functions, but it goes through the same functions for communications. Issues related to MPI arguments can be tested through other tools.

1.1 Motivating Examples

The MPI specification requires that all MPI processes call the same collective operations (blocking and non-blocking since MPI-3) in the same order [11]. These calls do not have to occur at the same line of source code, but the dynamic sequence of collectives should be the same otherwise a deadlock can occur. In addition, MPI calls should be cautiously located in multi-threaded regions. Focusing only on MPI, in Listing 1, because of the conditional in line 2 (`if` statement), some processes may call the `MPI_Reduce` function while others may not. Similarly, in Listing 2, some MPI ranks may perform a blocking barrier (`MPI_Barrier`) while others will call a non-blocking one (`MPI_Ibarrier`). The sequence is the same (call to one barrier), but this blocking/non-blocking matching is forbidden by the MPI specification.

Listing 1

```
1  void f(){
2    if(...)
3    {
4       #pragma omp parallel
5       {
6           #pragma omp single
7           {
8              MPI_Reduce(..)
9           }
10      }
11   }
12 }
```

Listing 2

```
1  void f(){
2    if(...)
3       MPI_Barrier(..)
4    else
5       MPI_Ibarrier(...)
6    #pragma omp parallel
7    {
8      /***/
9    }
10 }
```

Listing 3

```
1  void f(){
2    #pragma omp parallel
3    {
4      /***/
5      #pragma omp master
6      {
7         MPI_Send(..)
8         /***/
9      }
10   }
11 }
```

Listing 4

```
1  void f(){
2    #pragma omp parallel
3    {
4      #pragma omp single nowait
5      {
6        MPI_Reduce(..)
7      }
8      #pragma omp single
9      {
10       MPI_Reduce(..)
11     }
12   }
13 }
```

Fig. 1 MPI+OpenMP Examples with different uses of MPI calls.

Regarding hybrid MPI+OpenMP applications, the MPI API defines four levels of thread support to indicate how threads should interact with MPI: `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED` and `MPI_THREAD_MULTIPLE`. MPI processes can be multithreaded but the MPI standard specifies that "it is the user responsibility to prevent races when threads within the same application post conflicting communication calls" [11]. In Listing 2, MPI calls are executed outside the multithreaded region. This piece of code is therefore compliant with the `MPI_THREAD_SINGLE` level. But MPI communications may appear inside OpenMP blocks. For example, the MPI point-to-point function at line 7 in Listing 3 is inside a `master` block. The minimum thread level required for this code is therefore `MPI_THREAD_FUNNELED`. However, calls located inside a single or master block may lead to different thread support. Indeed, in Listing 4, two `MPI_Reduce` are in different `single` regions. Because of the `nowait` clause on the first `single` region, these calls are performed simultaneously by different threads. This example requires the maximum thread support level i.e., `MPI_THREAD_MULTIPLE`.

These simple examples illustrate the difficulty for a developer to ensure that MPI calls are correctly used inside an hybrid MPI+OpenMP application. A tool able to check, for each MPI call, in which thread context it can be performed would help the application developer to know which thread-level an application requires. Furthermore, beyond this support, checking deadlock of MPI collective communications in presence of OpenMP constructs can be very tricky. In this paper, we propose an extension of PARCOACH to tackle these issues, with the help of an interprocedural analysis to improve the compile-time detection.

Section 2 gives an overview of the PARCOACH platform with a description of its static and dynamic analyses for hybrid MPI+OpenMP applications. Then, Section 3 describes an interprocedural extension of the PARCOACH static pass. Section 4 presents experimental results and finally Section 5 concludes the paper.

2 PARCOACH Static and Dynamic Analyses for Hybrid Applications

PARCOACH uses a two-step method to verify MPI+OpenMP applications as shown in Figure 2. The first analysis is located in the middle of the compilation chain, where the code is represented as an intermediate form. Each function of a program is depicted by a graph representation called *Control Flow Graph* (CFG). PARCOACH analyses the CFG of each function to detect potential errors or deadlocks in a program. When a potential deadlock is detected, PARCOACH reports a warning with precise information about the possible deadlock (line and name of the guilty MPI communications, and line of conditionals responsible for the deadlock). Then the warnings are confirmed by a static instrumentation of the code. Note that whenever the compile-time analysis is able to statically prove the correctness of a function, no code is inserted in the program, reducing the impact of our transformation on the execution time. If deadlocks are about to occur at runtime, the program is stopped and PARCOACH returns error messages with compilation information.

This section describes the following new features of PARCOACH: (i) detection of the minimal MPI thread-level support required by an MPI+OpenMP application (see [9] for more details) and (ii) checking misuse of MPI blocking and nonblocking collectives in a multi-threaded context (extension of [10]).

2.1 MPI Thread-Level Checking

This analysis finds the right MPI thread-level support to be used and identifies code fragments that may prevent conformance to a given level. Verifying the compliance of an MPI thread level in MPI+OpenMP code resorts to check the placement of MPI calls. To determine the thread context in which MPI calls are performed, we augment the CFGs by marking the nodes containing MPI calls (point-to-point

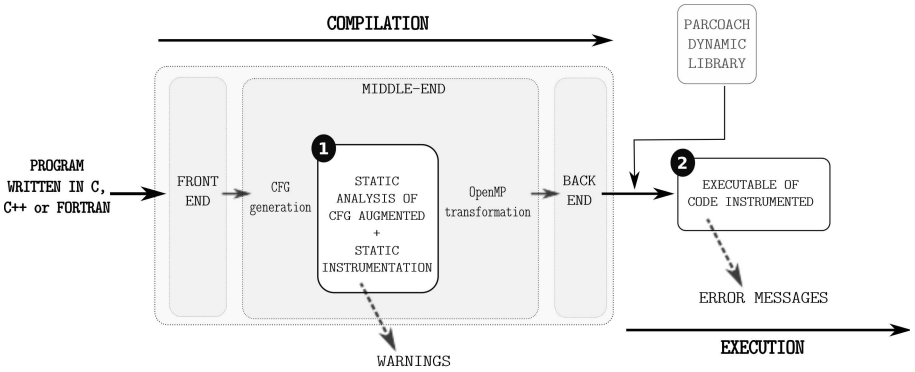


Fig. 2 PARCOACH two-step analysis overview

and collective). Then, with a depth-first search traversal, we associate a *parallelism word* to each node. As defined in [9], a parallelism word is the sequence of OpenMP parallel constructs (P:`parallel`, S:`single`, M:`master` and B:`barrier` for implicit and explicit barriers) surrounding a node from the beginning of the function to the node. The analysis detects CFG nodes containing MPI calls associated to parallelism words defining a multithreaded context and forbidden concurrent calls. Based on this analysis, the following section describes how collectives operations can be verified in a multithreaded context.

2.2 MPI Collective Communication Verification

This analysis proposes a solution to check the sequence of collective communications inside MPI+OpenMP programs. PARCOACH verifies that there is a total order between the MPI collective calls within each process and it ensures that this order is the same for all MPI ranks. Our analysis relies on checking 3 rules:

1. Within an MPI process, all collectives are executed in a monothreaded context;
2. Within an MPI process, two collective executions are sequentially ordered, either because they belong to the same monothreaded region or because they are separated by a thread synchronization (no concurrent monothreaded regions);
3. The sequence of collectives are the same for all MPI processes (i.e., sequences do not depend on the control flow).

A function is then said to be potentially statically incorrect if at least one of the three categories presented in Figure 3 is verified. This section describes how these error categories can be detected.

Category 1 Detection: This phase of the static analysis corresponds to the detection of MPI collectives that are not executed in a monothreaded region. To this end, we use the parallelism words defined in [10]. A parallelism word defines a

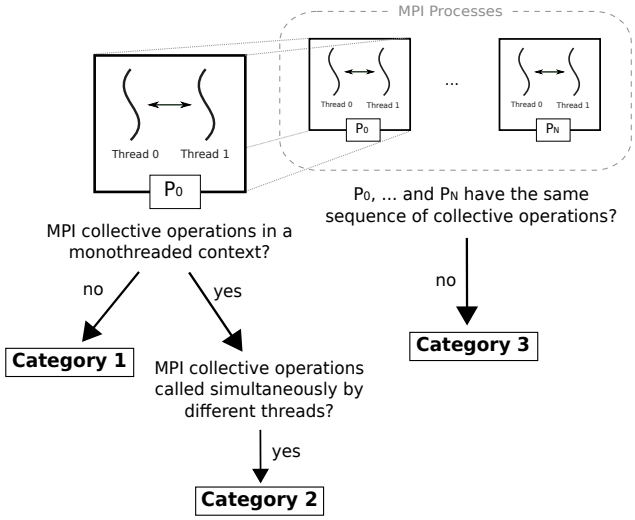


Fig. 3 Categories of possible errors in a hybrid program with N MPI processes and two threads per process.

monothreaded context if it ends with an S or an M (ignoring Bs). If the parallelism word has a sequence of two or more P with no S or M in-between, it implies the parallelism is nested. Even if the word ends with an S or M, one thread for each thread team can execute the MPI collectives.

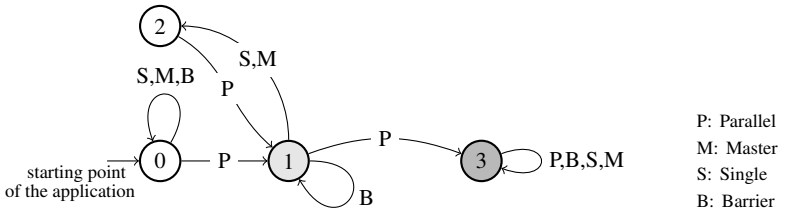


Fig. 4 Automata of possible parallelism words. Nodes 0 and 2 correspond to code executed by the master thread or a single thread. Node 1 corresponds to code executed in a parallel region, and 3 to code executed in nested parallel region.

For this part, it is not necessary to seperate `single` from `master` regions. So the finite-state automaton in [9] is simplified into the automaton presented Figure 4. It recognizes the language of parallelism words corresponding to monothreaded regions. States 0 and 2 are the accepting states and the language L defined by $L = (S|M|B|PB*S|PB*M)*$ contains the accepted words (parallelism words ending

by S or M without a repeated sequence of P).

Category 2 Detection: For this analysis, MPI collective operations are assumed to be called in monothreaded regions, as defined in the previous section. However, different MPI collectives can still be executed simultaneously if monothreaded regions are executed in parallel. This phase corresponds to the detection of MPI collective calls in concurrent monothreaded regions.

Two nodes n_1 and n_2 are said to be in concurrent monothreaded regions if they are in monothreaded regions and if their parallelism words $pw[n_1]$ and $pw[n_2]$ are respectively equal to wS^ju and wS^kv where w is a common prefix (possibly empty) with $j \neq k$, u and v words in $(P|S|B)*$.

Category 3 Detection: Once the sequence of MPI collective calls are verified in each MPI process, we must check that all sequences are the same for all processes. To verify that we rely on Algorithm 1 proposed in [7] with the extension of non-blocking collectives detailed in [4]. It detects MPI blocking and non-blocking collective mismatches by identifying conditionals potentially leading to a deadlock situation (set S). A warning is also issued for collective calls located in a loop as they can be called different times if the number of iterations is not the same for all MPI processes.

Algorithm 1 Step 1: Static Pass of hybrid programs

```
1: function HYBRID_STATIC_PASS(G = (V,E), L)
2:                          ▷ G: CFG, L: language of correct parallelism words
3:     DFS(G, entry(G))                          ▷ parallelism words construction
4:     MULTITHREADED_REGIONS(G, L)                          ▷ creates set S_ipw
5:     CONCURRENT_CALLS(G)                                  ▷ creates set S_cc
6:     STATIC_PASS(G)                                       ▷ creates set S
7: end function
```

Static Pass Algorithm: To wrap-up all static algorithms, Algorithm 1 shows how analyses are combined. First the *DFS* function creates parallelism words. Then *MULTITHREADED_REGIONS* and *CONCURRENT_CALLS* procedures respectively detect categories 1 and 2 of errors. Finally the *STATIC_PASS* procedure detects category 3 of errors.

2.2.1 Static Instrumentation

The compile-time verification outputs warnings for MPI collective operations that may lead to an error or deadlock. Nevertheless the static analysis could lead to false positives if the actual control-flow divergence is not happening during the execution. To deal with this issue, we present a dynamic instrumentation that verifies warnings emitted at compile-time.

Algorithm 2 Library Functions To Check MPI collectives

```
1: function CC_ipw                                  ▷ Detect collectives in multithreaded regions
2:     if pw_e ∉ L_e then
3:         MPI_ABORT(com, 0)
4:     end if
5: end function
6:
7: function CC_cc                                   ▷ Detect concurrent collective calls
8:     CC_ipw
9:     if collective_lock = 1 then
10:        MPI_ABORT(com, 0)
11:    else
12:        #pragma omp atomic write
13:        collective_lock = 1
14:    end if
15: end function
16:
17: function CC(com_c, i_c)                         ▷ Detect collective calls mismatches
```

To dynamically verify the total order of MPI collective sequences in each MPI process, validation functions (CCipw and CCcc) are inserted in nodes in the sets Sipw and Scc generated by the static pass (see Algorithm 1). These functions are depicted in Algorithm 2. Function CCipw detects incorrect execution parallelism words and Function CCcc detects concurrent collective calls. To dynamically verify the total order of MPI collective sequences between processes, a check collective function *CC* is inserted before each MPI collective operation and before `return` statements. *CC* is depicted in Algorithm 2 in [6]. It takes as input the communicator com_c related to the collective call c and a color i_c specific to the type of collective. As multiple threads may call *CC* before `return` statements, this function is wrapped into a `single` pragma. Each function of a program is instrumented by Algorithm 3. If an error is about to occur, the program is stopped and an error message is returned with error type information.

3 Interprocedural Analysis

Because PARCOACH relies on an intraprocedural analysis, it miss errors across function boundaries and therefore it may produce false positive as well as false negative results. To extend PARCOACH with an interprocedural mechanism, we extended the intraprocedural approach through the application Call Graph (CG): nodes represent functions and edges model possible calls.

The main idea is to compute and reuse the summaries of each CFG through a CG traversal in reverse invocation order. For this purpose, the intraprocedural analysis is modified to return the valid sequence of collective operations for each function (*ValidSeq*). Based on this sequence of collective operations that all MPI processes

Algorithm 3 Step 2: Selective Static Instrumentation

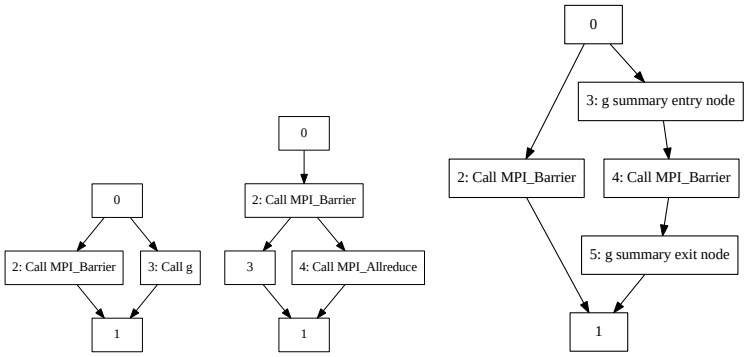
```
1: function INSTRUMENTATION(communicator, G, S, S_ipw, S_cc)
2:                                  ▷ G: CFG, S, S_ipw, S_cc: sets created at compile-time
3:     if S ∪ S_ipw ∪ S_cc ≠ ∅ then
4:         ** STEP 1: Control flow errors detection **
5:         for n in nodes containing a call to collective c do
6:             Insert call to CC(com_c, i_c) before the call to c
7:         end for
8:         Before return statements insert
9:             # pragma omp single
10:                CC(communicator, 0) }
11:        ** STEP 2: Collectives in multithreaded regions **
12:        for n ∈ S_ipw do
13:            Insert call to CC_ipw() as the first statement of n
14:        end for
15:        ** STEP 3: Concurrent MPI calls detection **
16:        for n ∈ S_cc do
17:            Insert call to CC_cc() as the first statement of n
18:            Insert collective_lock = 0 after the barrier(s) successors of the region created by n
19:        end for
20:    end if
21: end function
```

Algorithm 4 Interprocedural Analysis

```
1: function INTERPROCEDURAL_ANALYSIS(⋃_f CFG_f, CG)          ▷ CG: Callgraph
2:     Seq ← {}, O ← ∅
3:     for each n ∈ CG in reverse topological order do
4:         n.ValidSeq ← {} , O_n ← ∅
5:         for each f ∈ SUCC_CG(n) do
6:             Replace f in n by f.ValidSeq
7:         end for
8:         (O_n, ValidSeq) ← INTRAPROCEDURAL_ANALYSIS(CFG_n)
9:         n.ValidSeq ← ValidSeq, O ← O ∪ O_n
10:    end for
11:    return O
12: end function
```

will encounter, Algorithm 4 presents the interprocedural analysis. It takes as input the CG of a program and all the CFGs and returns the set O of conditionals that can lead to a deadlock.

Figure 5(a) shows the `main` CFG calling function `g` (whose CFG is depicted in Figure 5(b)). Performing our intraprocedural analysis would lead to a deadlock warning on `main` (for the `MPI_Barrier` operations) and on `g` (for the `MPI_Allreduce` collective). The resulting CFG after applying Algorithm 4 is illustrated in Figure 5(c): the call to `g` is replaced by the sequence of collective executed in `g`. Then, simply invoking the intraprocedural analysis on this new CFG results in no warning because there is one call to `MPI_Barrier` on each path of the program. Therefore, the only warning issued by our combined analysis is related to the call to `MPI_Allreduce` in function `g` (related to the `if` statement in node 2).



(a) CFG of main (b) CFG of g (c) Program CFG

Fig. 5 Example of interprocedural analysis

4 Experimental Results

We extended the PARCOACH implementation (GCC 4.7 plugin) to add analysis of hybrid applications. Thus, it is associated to GCC but simple to deploy in existing environments as it does not modify the compilation chain. To show the impact of PARCOACH analysis on the compilation and execution time, we tested the NAS-MZ [12], AMG benchmark [1], the EPCC suite [2] and HERA [5]. All results were conducted on Tera100, a petaflopic supercomputer at the CEA.

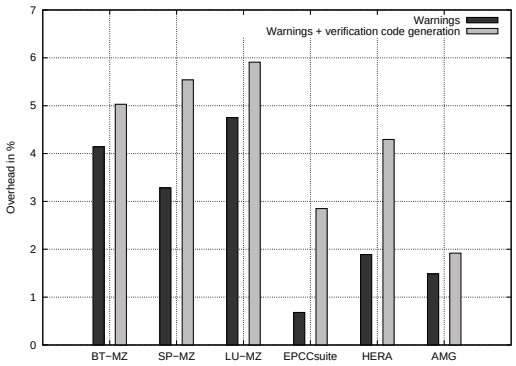
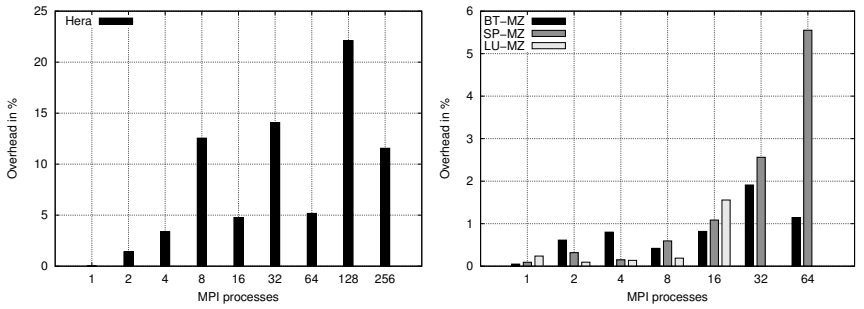


Fig. 6 Overhead of average compilation time with and without verification code generation

Figure 6 displays the overhead of compiling the applications with PARCOACH (only with our static analysis, or with the analysis and the static instrumentation). It shows that PARCOACH introduces a low compilation overhead (under 6%). The execution time overheads obtained for the NAS benchmarks and HERA are pre-

sented in Figure 7 running on MPICH with GCC OpenMP. The overheads obtained are under 25% which is reasonable for debugging purpose.



(a) Execution-Time Overhead for HERA (b) Execution-Time Overhead for NASPB-MZ

Fig. 7 Execution-Time Overhead for MZ (NAS class B) and HERA with 8 threads per MPI process (Strong scaling)

```
int RestartIO_GLEAN :: Close (void) {
    int status = -1;
    if (m_mode == WRITE_CHECKPOINT)
    {
        switch (m_interface)
        {
            case USE_POSIX:
                status = this->__POSIX_Close_Checkpoint(); break;
            case USE_MPIIO:
                status = this->__MPIIO_Close_Checkpoint(); break;
        }
    }
    else if (m_mode == READ_RESTART)
```

(a)

```
int RestartIO_GLEAN :: __MPIIO_Close_Checkpoint (void)
{
    [...]
    MPI_Barrier(m_partitionComm);
}
```

(b)

```
int RestartIO_GLEAN :: __POSIX_Close_Checkpoint (void)
{
    [...]
    MPI_Barrier(m_partitionComm);
    [...]
    MPI_Barrier(m_partitionComm);
    [...]
}
```

(c)

Fig. 8 Pieces of HACC/IO module code

The interprocedural analysis has also been implemented and integrated in PARCOACH (as a GCC plugin combined with a Python script). Figure 8 shows pieces of code from the IO module of the CORAL benchmark HACC. The calls in Figures 8(b) and 8(c) contain one and two calls to an MPI collective, respectively. Hence when the function in Figure 8(a) calls the others in different paths because of the switch, the execution could deadlock if the processes follow different paths. Of course the conditional statement does not depend on the rank number of each process, and therefore this is just a false positive. This interprocedural must be extended to a data flow analysis with the aim to study the dependence of these condition variables in order to know if they depend on the process rank or not.

5 Conclusion

The MPI+OpenMP approach is one solution to tackle the increasing node-level parallelism and the decreasing amount of memory per compute unit. Some production codes are already hybrid and other applications are in the development process. It is driven by available tools that could help debugging. That is why we developed the platform PARCOACH that helps application developers to check which interaction support is required for a specific hybrid code and checks the correct usage of blocking and non-blocking MPI collective communications in an MPI+OpenMP application. The main advantage of PARCOACH is that it highlights the statements responsible for the execution path potentially leading to future deadlocks or unspecified behaviors. We propose an adaptation of PARCOACH analyses to an interprocedural analysis. This enables us to reduce the number of false positives returned by the initial static analysis. However, this interprocedural analysis could be improved to propagate collective issue information and can be coupled to a data-flow analysis to avoid false positive results.

References

1. CORAL Benchmarks. https://asc.llnl.gov/CORAL-benchmarks/
2. Bull, J.M., Enright, J.P., Guo, X., Maynard, C., Reid, F.: Performance Evaluation of Mixed-Mode OpenMP/MPI Implementations. Intl. J. of Parallel Programming **38**(5-6), 396–417 (2010)
3. Hilbrich, T., Müller, M.S., Krammer, B.: Detection of Violations to the MPI Standard in Hybrid OpenMP/MPI Applications. In: Intl. Conf. on OpenMP in a New Era of Parallelism, pp. 26–35. Springer-Verlag (2008)
4. Jaeger, J., Saillard, E., Carribault, P., Barthou, D.: Correctness Analysis of MPI-3 Non-Blocking Communications in PARCOACH. In: Proceedings of the 22Nd European MPI Users' Group Meeting, EuroMPI '15, pp. 16:1–16:2. ACM (2015)
5. Jourdren, H.: HERA: A hydrodynamic AMR Platform for Multi-Physics Simulations. In: T. Plewa, T. Linde, V.G. Weirs (eds.) Adaptive Mesh Refinement - Theory and Applications, pp. 283–294. Springer (2003)
6. Saillard, E., Carribault, P., Barthou, D.: Combining Static and Dynamic Validation of MPI Collective Communications. In: Proceedings of the European MPI Users' Group Meeting, EuroMPI'13, pp. 117–122. ACM (2013)
7. Saillard, E., Carribault, P., Barthou, D.: PARCOACH: Combining static and dynamic validation of MPI collective communications. International Journal of High Performance Computing Applications (2014)
8. Saillard, E., Carribault, P., Barthou, D.: Static Validation of Barriers and Worksharing Constructs in OpenMP Applications. In: International Workshop on OpenMP, pp. 73 – 86 (2014)
9. Saillard, E., Carribault, P., Barthou, D.: MPI Thread-Level Checking for MPI+OpenMP Applications. In: EuroPar (2015)
10. Saillard, E., Carribault, P., Barthou, D.: Static/Dynamic Validation of MPI Collective Communications in Multi-threaded Context. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pp. 279–280. ACM (2015)
11. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1, June 2015. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
12. NASPB site: http://www.nas.nasa.gov/software/NPB